

Copying and Moving Workshop Solutions

- The code for the solutions is in a zip file which is available as a separate downloadable resource

Can you finish this for me...?

- `std::thread` is a move-only type in the Standard Library
 - The details of what it does are not important for this exercise
- The code on the next slide is a partially-completed RAI class for managing `std::thread` objects
 - The programmer responsible for it has gone on holiday
 - However, he has written the destructor. He assures us this is the only non-obvious part of the code
 - He has also left instructions that the class should be move-only and not copyable

Can you finish this for me...?

```
#include <thread>

class thread_guard {
    std::thread t;
public:
    ~thread_guard() {
        if (t.joinable()) {
            t.join();
        }
    }
};
```

Can you finish this for me...?

- Complete the class by adding constructors and assignment operators as necessary
- Write a program to test your class
- Check that it supports move operations but not copy operations
- (Depending on your environment, you may need to link against a thread library to get this to compile. Please ask if uncertain!)

Constructor

- We know that `std::thread` is move-only, so this will have to be a move constructor

```
thread_guard(std::thread&& t) : t(std::move(t)) {}
```

- We will make the constructor explicit, to avoid conversions
 - This is unlikely to be a problem here, but is good practice

```
explicit thread_guard(std::thread&& t) : t(std::move(t)) {}
```

- Since we have defined a constructor, the compiler will not generate a default constructor for us

```
explicit thread_guard() = default;
```

// Not required, but may be useful

Copy constructors

- The class must not be copyable, so we prevent the compiler from generating copy operators

```
thread_guard(const thread_guard&) = delete;  
thread_guard& operator=(const thread_guard&) = delete;
```

- The class must be moveable
- Since we have defined a destructor, the compiler will not generate move operators, so we must add them

```
thread_guard(thread_guard&&) noexcept = default;  
thread_guard& operator=(thread_guard&&) noexcept = default;
```

Test Program

```
int main() {  
    std::thread t;  
    thread_guard tg{std::move(t)};           // Compiles  
    thread_guard tg2;                         // Compiles  
    // tg2 = tg;                             // Does not compile  
    // thread_guard tg3(tg);                  // Does not compile  
    thread_guard tg4(std::move(tg));          // Compiles  
    std::thread t2;  
    thread_guard tg5 = thread_guard(std::move(t2)); // Compiles  
}
```


- We need to deduce the interface of the Camera class
 - We know that it has public member functions `take_picture()` and `replace()`
 - We know that it has a constructor that takes a pointer to `MemoryCard`, and `Camera` has custody of this pointer
 - `Camera` needs to implement the "rule of five" member functions since it is managing a resource
 - `Camera` is moveable but not copyable. This allows it to maintain custody of the pointer
 - To make it uncopyable, the copy operators must be deleted

Camera class

```
class Camera {  
    private:  
        MemoryCard *m;  
    public:  
        Camera(MemoryCard *m) : m(m) {}  
        Camera(const Camera&) = delete;  
        Camera(Camera&& other) noexcept;  
        Camera& operator=(const Camera&) = delete;  
        Camera& operator=(Camera&& other) noexcept;  
        ~Camera();  
        void take_picture();  
        void replace(MemoryCard *);  
};
```